Advanced Review

Sparse matrix computations with application to solve system of nonlinear equations

Shahadat Hossain[1] and Trond Steihaug[2]*

Numerical linear algebra is an essential ingredient in algorithms for solving problems in optimization, nonlinear equations, and differential equations. Spanning diverse application areas, from economic planning to complex network analysis, modeling and solving problems arising in those areas share a common theme: numerical calculations on matrices that are sparse or structured or both. Linear algebraic calculations involving sparse matrices of order 10^9 are now routine. In this article, we give an overview of scientific calculations where effective utilization of properties such as sparsity, problem structure, etc. play a vital role and where the linear algebraic calculations are much more complex than their dense counterpart. This is partly because operation and storage involving known zeros must be avoided, and partly because the fact that modern computing hardware may not be amenable to the specialized techniques needed for sparse problems. We focus on sparse calculations arising in nonlinear equation solving using the Newton method. © 2013 Wiley Periodicals, Inc.

How to cite this article:
WIREs Comput Stat 2013, 5:372–386. doi: 10.1002/wics.1269

Keywords: optimization; Newton's method; sparse linear algebra; sparse matrix storage; Jacobian matrix

INTRODUCTION

The notion of 'sparsity', while widely appreciated, is difficult to define. Sparsity exploiting numerical methods are expected to avoid storing the elements known to be zero or doing arithmetic operations with them. Efficiency of many sparse operations depends on the number of zero elements as well as the distribution of zero and nonzero elements. An informal working definition of a sparse matrix according to Wilkinson can be phrased as 'any matrix with enough zeros that it is computationally advantageous to exploit them'.[1] Frequently, large-scale problems contain useful 'structures' which may or may not be explicitly displayed. Some 'preprocessing' of the problem may be needed to enable structures in exploitable form. Sparse matrix algorithms often require special techniques that are quite distinct from their dense counterpart. Computational complexity of sparse matrix operations is affected by factors such as memory traffic and the size and organization of cache memory, in addition to the number of floating-point operations. Modern off-the-shelf uniprocessor computing systems usually contain several levels of cache memories lodged between the Random Access Memory (RAM) and the Central Processing Unit (CPU) registers (see Ref 2 for details on hierarchical memory systems and the related issues). In general, closer the cache to the CPU faster the time to access (and smaller the amount of data it can store) and transfer data to the CPU. Faster cache memories are usually more expensive. When a piece of data is needed by the CPU, the cache memory is searched by level. A *cache-miss* occurs when the requested data is not found at the current cache level, necessitating

*Correspondence to: Trond.Steihaug@ii.uib.no

[1] Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, Alberta, Canada

[2] Department of Informatics, University of Bergen, Bergen, Norway

Conflict of interest: The authors have declared no conflicts of interest for this article.

read and transfer of a 'block' of data (from the slower cache) that contains the requested data. To simplify discussion, we consider only one level of cache memory between the CPU and the RAM. For large-scale problems it is usually the case that the faster cache memory is not large enough to hold the input data in its entirety. The term *locality of reference* is used to signify the access pattern of data during the execution of an algorithm in a hierarchical memory computing system. The principle of data locality postulates that 'recently accessed data (temporal) and nearby data (spatial) are likely to be accessed in the near future'. Generally, better the reference locality for data fewer the cache-misses. As an example, consider scanning an array of length n holding integer values. Suppose a data read between the two memories (a slow RAM and a fast cache) corresponds to b bytes of data transfer. Then, assuming that each integer is encoded in four bytes, scanning the array will incur $4n/b$ compulsory cache misses and this number is essentially independent of the size of the cache. We say that scan operation has full spatial locality.

This article is structured as follows. First, we present two standard storage schemes for sparse matrices. Our focus is on data structures implementing these storage schemes for matrices with 'general sparsity pattern'. The discussion is centered around a small subset of kernel operations from sparse '*B*asic *L*inear *A*lgebra *S*ubprograms (BLAS)' specification.[3] With regard to computer implementation we assume a single-processor single-core hierarchical memory system; no specific hardware configuration is implied. The algorithmic description for sparse operations is provided as pseudocode with sufficient details such that the conversion to computer code in a high-level language is straight-forward. Next, we present the $\boldsymbol{LU}$-factorization of a sparse matrix via Gaussian elimination. The high-level description of the algorithm emphasizes vector/matrix operations and sparsity. This also sets the stage for the application of sparse techniques in Newton's method for solving systems of nonlinear equations where the major computational tasks (at each iteration) are the determination of the Jacobian matrix and the solution of the associated linear system. The Curtis, Powell, and Reid method,[4] the CPR method, for the determination of a sparse Jacobian matrix is explained. In the second part of this section we briefly outline stationary iterative methods. The section concludes with the inexact Newton methods (truncated Newton) where the linear system is solved only approximately using an iterative method. In the final section we summarize this review.

Our main notational conventions are as follows. A matrix is denoted by an uppercase letter (bold) while a vector is denoted by a lowercase letter (bold). A (real) scalar is usually denoted by a Greek letter. Unless stated otherwise, all vectors are column vectors. In a displayed matrix, a 0 or a blank symbol represents a vanishing (zero) entry; any other mark represents a nonzero. We use the 'colon notation'[5] to specify part of a matrix: $\boldsymbol{A}(i_1 : i_2, j_1 : j_2)$ refers to the elements with row indices in $\{i_1, i_1 + 1, \dots, i_2\}$ and column indices in $\{j_1, j_1 + 1, \dots, j_2\}$. Then, the (i, j)th element is obtained as $i_1 = i_2 = i$ and $j_1 = j_2 = j$ and denoted by $\boldsymbol{A}(i, j)$ which can be simplified to a_{ij}. For a vector of indices $\boldsymbol{v}$, $\boldsymbol{A}(\boldsymbol{v},:)$ and $\boldsymbol{A}(:,\boldsymbol{v})$, respectively refers to rows and columns whose indices are in $\boldsymbol{v}$. A set of indices or index pairs is often denoted by a calligraphic letter. The sparsity pattern or sparsity structure of $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is denoted by $\mathcal{S}(\boldsymbol{A}) = \{(i, j)\,|a_{ij} \neq 0\}$. Likewise, the sparsity pattern of $\boldsymbol{x} \in \mathbb{R}^n$ is denoted by $\mathcal{S}(\boldsymbol{x}) = \{i|\boldsymbol{x}(i) \neq 0\}$. The ith coordinate vector (ith row or column of the identity matrix $\boldsymbol{I}$) is denoted by $\mathbf{e}_i$. For brevity, we usually omit the qualifier 'sparse' from the term 'sparse matrix'. In the pseudocode description and other relevant places the form *variable* ← *expression* indicates the assignment of the value of expression to variable and usually involves memory traffic.

DATA STRUCTURES FOR SPARSE MATRICES

Barrett et al.[6] and Saad[7] give an overview of storage schemes for both structured and general sparse matrices arising in iterative methods. Duff et al.[8] in their book on direct methods devote a chapter on data structures and elementary linear algebra operations. An example of a modern C implementation of data structures and sparse linear algebra operations can be found in Ref [9]. An extensible object-oriented implementation of sparse matrix storage and operations that utilize *design patterns* can be found in Ref [10].

The choice of a storage scheme for sparse matrices with no specific 'structure' is influenced by a number of factors including the computing system (hierarchical memory, multicore, vector or superscalar, course grain, or fine grain parallelism) and the operations to be supported. There is no one storage scheme that will yield the optimum result in all situations. From the computer implementation view point linear algebraic operations of interest can be categorized into two: ones which simply access the matrix elements (read-only) and the ones which alter the matrix in some way (read–write).[a] Multiplying a matrix by a vector is a read-only task; $\boldsymbol{LU}$-factorization of a square matrix usually leads to 'fill-ins'—an example of a read–write operation in which a nonzero element is created at a location

(a) **(b)**

$$\begin{bmatrix} 0 & a_{12} & 0 & 0 & 0 \\ a_{21} & 0 & 0 & a_{24} & 0 \\ 0 & 0 & a_{33} & 0 & a_{35} \\ 0 & a_{42} & 0 & 0 & a_{45} \\ a_{51} & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \Longrightarrow \begin{array}{l} \{(4,2,a_{42}), (5,1,a_{51}), (4,5,a_{45}), (5,4,a_{54}), (1,2,a_{12}), \\ (3,3,a_{33}), (2,4,a_{24}), (3,5,a_{35}), (2,1,a_{21}), (5,5,a_{55})\} \end{array}$$

FIGURE 1 | Sparsity pattern.

$m = 5$	*rowind*	4	5	4	5	1	3	2	3	2	5
$n = 5$	*colind*	2	1	5	4	2	3	4	5	1	5
$nnz = 10$	*value*	a_{42}	a_{51}	a_{45}	a_{54}	a_{12}	a_{33}	a_{24}	a_{35}	a_{21}	a_{55}

FIGURE 2 | Coordinate storage scheme.

that was zero. Analyzing computational performance of sparse matrix operations is much more complex than their dense counterpart: simple floating point operations (flops) count may not give the true picture. Combinatorial computations, e.g. column reordering, have very low computational intensity (ratio of flops count to memory access); the running time is dominated by the time to access the nonzero elements. For the sparse matrices and dense vectors considered in this section we assume that the number of nonzero elements *nnz* and the vector dimensions n, m are larger than the available cache storage.

Our discussion of technical issues related to data structure selection for sparse matrices is done in reference to a set of kernel linear algebraic operations that are fundamental to sparse computations arising in optimization. Sparse BLAS[3] is a specification which, like its dense counterpart, is widely accepted as the standard. In this article, we mainly focus on the following operations:

1 SpDot: $p \leftarrow \boldsymbol{x}^\top \boldsymbol{y}$
2 SpAxpy: $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$
3 SpMv: $\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{x}$
4 SpMM: $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}$

In addition to the above operations we discuss the utility operations *gather*, *gather and zero* (SpGa, SpGaZ), and *scatter* (SpSc). Computations in finite-difference (FD) discretization of a class of partial differential equations often yield matrices with 'regular sparsity' pattern, e.g. band matrices with limited bandwidth.[7] Such information can be utilized in designing effective data structure for specific patterns.[6] In this article, we mainly assume 'general sparsity pattern', as opposed to 'regular pattern'.

Coordinate Storage

Figure 1 displays sparse matrix $\boldsymbol{A}$: a two-dimensional array where zeros are explicitly shown, as a collection of 3-tuples (i, j, a_{ij}) representing only nonzero entries, where i denotes the row index, and j denotes the column index. Unless otherwise noted, the indices (row, column, vector, or array component) begin at the value of 1. The collection of 3-tuples in Figure 1 constitutes one of the simplest storage scheme, the *Coordinate Storage (COS)* scheme, for sparse matrices. An array-based data structure for COS can be given using a pair of arrays, *rowind*, *colind*, to store the index pairs (i, j) and an array *value* to store the associated nonzero numerical values. The COS data structure for matrix $\boldsymbol{A}$ shown in Figure 2 provides a complete description: n, m, and *nnz* denote the number of columns, rows, and nonzero elements, respectively. The nonzero elements in array *value* shown symbolically are stored in memory in no specific order. Therefore, the data structure setup in COS is straightforward—obtain n, m, and *nnz*, and record each nonzero entry into the arrays in the order they are made available (e.g., as a text file). Unfortunately, except for a few kernel operations, the COS data structure is inefficient. On the other hand, its inherent simplicity led to its adoption as a standard format for exchange of sparse matrices.[11]

Compressed Row (Column)

Compressed Sparse Row (CSR) can be considered as a *de facto* base-line storage scheme for general sparse matrices in that more specialized and sophisticated storage schemes are defined by extending it.

Figure 3(b) shows matrix $\boldsymbol{A}$ (from Figure 1) represented in CSR scheme using three arrays *rowptr*, *colind*, and *value*. Each row of $\boldsymbol{A}$ is stored as a compressed sparse vector. A sparse vector can be stored in compressed form using two arrays: one array to hold the indices of the nonzero entries and the other to hold the corresponding nonzero values. Thus, the CSR storage scheme can be viewed as a natural extension of the compressed sparse vector storage

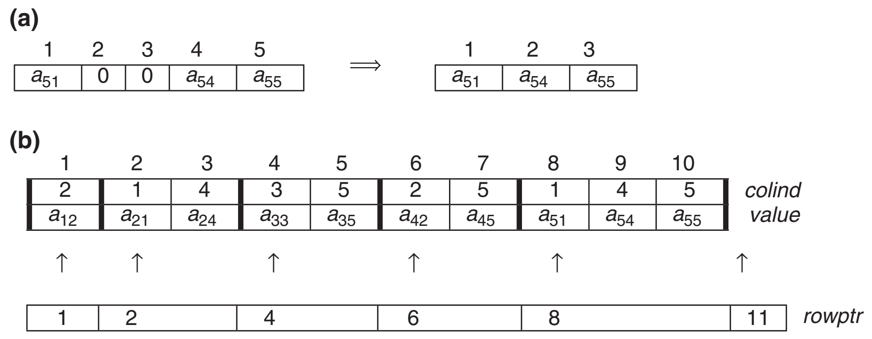

(a)
1 2 3 4 5
a_{51} 0 0 a_{54} a_{55}
$\Longrightarrow$
1 2 3
a_{51} a_{54} a_{55}
(b)
1 2 3 4 5 6 7 8 9 10
2 1 4 3 5 2 5 1 4 5
a_{12} a_{21} a_{24} a_{33} a_{35} a_{42} a_{45} a_{51} a_{54} a_{55}
colind
value
1 2 4 6 8 11
rowptr

FIGURE 3 | Compressed sparse row storage scheme.

where the sparse row vectors are stored contiguously. Figure 3(a) shows the sparse vector corresponding to row 5 (on left) and its compressed form (on right). In CSR, rows are stored in increasing order of row indices. In the figure nonzero elements in each row are sorted by increasing column indices although this is not strictly necessary. Array *rowptr* provides access to the elements of the sparse matrix such that *rowptr*(i) indicates the location (i.e., index) of the first element of row i in arrays colind and value. For example, *rowptr*(5) is 8 indicating that the first nonzero element of row 5 is located at (5, *colind*(8)) having value *value*(8) $\equiv a_{51}$. Thus, elments in row i are accessed as

$$value\left(k\right) \text{ and are located in } \left(i, colind\left(k\right)\right),$$
$$k = rowptr\left(i\right) \quad \text{to} \quad rowptr\left(i+1\right)-1.$$

The *Compressed Sparse Column (CSC)* is simply the transposed matrix stored using the CSR. Sparse matrices in Harwell–Boeing collection[12] are given in CSC. The MATLAB computing environment[1] and CSparse[9] software use CSC representation for sparse matrices and the associated operations. The *Java Sparse Array (JSA)* proposed by Gundersen and Steihaug[13] is a flexible and dynamic storage scheme for implementing sparse matrix data type in object-oriented languages such as Java and C#. The JSA storage scheme has been found to be highly competitive with CSR scheme in terms of running time especially on sparse matrix algorithms where the sparsity of the result is not *a priori* known.

Sparse Matrix Operations

Let $\boldsymbol{x}$ be a sparse vector specified by arrays *xindex* and *xvalue* and let $\boldsymbol{y}$ be a dense vector given in array y (when there is no ambiguity, a vector and its array representation may be used interchangeably). The sparse gather operation SpGa corresponds to performing

$$xvalue \leftarrow y\left(xindex\right).$$

For example, with

1	2	3	
1.3	2.1	3.4	$xvalue$

1	2	3	
1	4	5	$xindex$

1	2	3	4	5	
1.0	1.0	2.0	3.0	5.0	y

an application of SpGa changes *xvalue* to

1	2	3
1.0	3.0	5.0

.

The sparse scatter operation SpSca performs the reverse of SpGa:

$$y\left(xindex\right) \leftarrow xvalue.$$

It is to be emphasized that the computational effort in implementing gather and scatter operations should be proportional to the number of nonzero entries, and not the length of the full vector.

SpDot computes and returns the inner product $p \leftarrow \boldsymbol{x}^{\top}\boldsymbol{y}$ for sparse vectors (packed vectors in Ref 8) $\boldsymbol{x}$ and $\boldsymbol{y}$ which are provided to the algorithm via arrays *xindex*, *xvalue*, and *yindex*, *yvalue*, respectively. The algorithm SpDot assumes that the nonzero entries are given in increasing order of their component index, e.g. as in Figure 3. The variables *nnzx* and *nnzy*, respectively, denote the length of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ obtained via a function called size (e.g., in MATLAB). The integer pointers i and j index into the arrays *xindex* and *yindex* and perform the multiplication only if they have identical values. In the worst-case all of *xindex* and *yindex* are accessed (e.g., *xindex* is 1, 3, 5, 7 and *yindex* is 2, 4, 6, 8) giving O(*nnzx* + *nnzy*) cost. The elements in each of the sparse vectors are accessed in order of increasing index yielding full spatial locality.

```
SpDot($xindex, xvalue, yindex, yvalue$)
$p \leftarrow 0.0$;
$nnzx \leftarrow$ size($xindex$); $nnzy \leftarrow$ size($yindex$);
$i \leftarrow 1; j \leftarrow 1$;
while (($i \le nnzx$) and ($j \le nnzy$))
    do
        if ($xindex(i) = yindex(j)$)
          then
                $p \leftarrow p + xvalue(i) * yvalue(j)$;
                $i \leftarrow i + 1; j \leftarrow j + 1$;
        elseif $xindex(i) < yindex(j)$
          then $i \leftarrow i + 1$;
        else
                $j \leftarrow j + 1$;
return $p$
```

```
SpDot-w($w, xindex, xvalue$)
$p \leftarrow 0.0$;
$nnzx \leftarrow$ size($xindex$);
for ($i \leftarrow 1$ to $nnzx$)
    do
        $p \leftarrow p + xvalue(i) * w(xindex(i))$;
return $p$
```

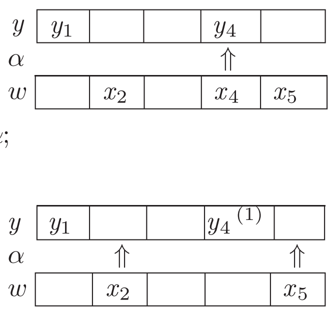
When the elements of the sparse vectors are not ordered, one can use a full-length work array w as shown in algorithm SpDot-w. In the algorithm work array w holds vector $\boldsymbol{y}$ in expanded form (using a scatter). The algorithm performs $O(nnzx)$ work. But access to $\boldsymbol{y}$ via array w may incur, in the worst-case, one cache miss per accessed element.

Multiplying a sparse matrix $\boldsymbol{A}$ by a dense vector $\boldsymbol{x}$ on the right, SpMv, results in a dense vector $\boldsymbol{y}$ and can be computed as

$$\text{for}\,(i \leftarrow 1 \ \text{to} \ m) \quad \boldsymbol{y}(i) \leftarrow \boldsymbol{y}(i) + \boldsymbol{A}(i,:)\,\boldsymbol{x},$$

with $\boldsymbol{y}$ initialized to zero at the beginning. The product $\boldsymbol{A}(i,:)\ \boldsymbol{x}$ is obtained by a minor modification of SpDot-w. If the elements in each row of matrix $\boldsymbol{A}$ are accessed in order, then the algorithm SpMv is expected to achieve full spatial locality in accessing matrix $\boldsymbol{A}$ and the result vector $\boldsymbol{y}$. Unfortunately, access to the multiplying vector $\boldsymbol{x}$ is dependent on the sparsity pattern of the rows of $\boldsymbol{A}$. As noted in the preceding paragraph, the elements of the dense vector $\boldsymbol{x}$ being accessed in arbitrary order for each row of the sparse matrix in SpMv, the number of cache misses can be as high as $O(nnz)$. The problem of improving data locality in SpMv has been studied extensively. Some of the techniques suggested by different researchers are column and/or row ordering,[14–16] loop unrolling, register blocking, cache blocking[17], etc.

```
SpAXPY($w, xindex, xvalue, yindex, yvalue, \alpha$)
$nnzx \leftarrow$ size($xindex$); $nnzy \leftarrow$ size($yindex$);
for ($i \leftarrow 1$ to $nnzx$)
    do
        $w(xindex(i)) \leftarrow xvalue(i)$;
for ($i \leftarrow 1$ to $nnzy$)
    do
        if ($w(yindex(i)) \neq 0$)
          then
                $yvalue(i) \leftarrow yvalue(i) + w(yindex(i)) * \alpha$;
                $w(yindex(i)) \leftarrow 0$;
for ($i \leftarrow 1$ to $nnzx$)
    do
        if ($w(xindex(i)) \neq 0$)
          then
                $nnzy \leftarrow nnzy + 1$;
                $yvalue(nnzy) \leftarrow \alpha * w(xindex(i))$;
                $w(xindex(i)) \leftarrow 0$;
                $yindex(nnzy) \leftarrow xindex(i)$;
```



Algorithm SPAXPY performs $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$ where x and y are sparse vectors and α is a scalar. This is an essential kernel operation in sparse Gaussian elimination and the algorithm demonstrates implementation features that are applicable to many other sparsity exploiting linear algebra routines. The work array w is input to the algorithm initialized to zero and when the algorithm returns, the vector is reinitialized to zero. Lines 2–4 essentially performs a scatter to load the sparse vector x into w at the cost of $O(nnzx)$. Lines 5–10 iterate over $nnzy$ elements of vector y and compute $y(i) \leftarrow y(i) + \alpha x(i)$ for each $y(i) \neq 0$ resulting in $O(nnzy)$ work. In the example (right figure), the only index i for which $x(i) \neq 0$, $y(i) \neq 0$ is $i = 4$. Lines 11–18 result in the 'fill entries' $\alpha x(i)$ for index i where $x(i) \neq 0$ while $y(i)$ is zero and therefore not stored in the compressed vector *yvalue*. This requires that either sparse vectors *yvalue* and *yindex* have sufficient 'elbow room' to accommodate the fill-ins or that the arrays are created on the heap memory[2] allowing them to be dynamically resized. Note that the array w is reset to zero at the conclusion of the algorithm. Usually, the fill-ins are inserted at the end of the result vector thereby destroying any order of the original elements. The example beside the algorithm SPAXPY shows the sparse vectors in full-length arrays. Entries at indices 2 and 5 are newly created and are to be inserted at the end. In SPAXPY and similar operations it is possible to create a zero due to numerical cancelation. In this article, we ignore numerical cancelation and a zero created in this way will be explicitly stored.

It is common to use an upper bound on the size of the result (with fill-in), e.g. smaller of $(nnzx + nnzy)$ and the full length n. MATLAB preallocates memory based on this kind of upper bound and leaves some unused memory if the actual number of entries is less than the precomputed upper bound.[1] MATLAB implements the work vector as a data structure called 'sparse accumulator'[1] which consists of one Boolean array, one dense array (w) and the sparse array containing the indices of the elements in w. For a sparse matrix calculation, it uses the sparse accumulator to compute the output one column at a time. The accumulator is initialized once at the beginning and repeatedly used for each column (as in SPGA). Accessing an arbitrary element of the current column is then constant-time. When the indices are not ordered, a linear time sorting algorithm, e.g. number sort or transposition sort is used to order them once and for all.

Determining Sparsity

In the matrix–matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ where both $\boldsymbol{A}$ and $\boldsymbol{B}$ are sparse, there are two main computational issues that need to be addressed. First, it is usually not known *a priori* the sparsity structure of the result matrix $\mathbf{C}$. Therefore, setting up the actual data structure for $\mathbf{C}$ is nontrivial. Second, in performing the multiplication $\boldsymbol{AB}$, one must take into account the data structure used to represent matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. If $c_{ij} \neq 0$ then

$$\mathbf{C}(i,j) = \sum_{\{k | k \in \mathcal{S}(\boldsymbol{A}(i,:)) \cap \mathcal{S}(\boldsymbol{B}(:,j))\}} \boldsymbol{A}(i,k)\, \boldsymbol{B}(k,j)$$

can be obtained using SPDOT-W requiring access to row i of $\boldsymbol{A}$ and column j of $\boldsymbol{B}$. If both $\boldsymbol{A}$ and $\boldsymbol{B}$ are represented in CSR scheme, accessing column j of $\boldsymbol{B}$ will be expensive. One can avoid column access by reorganizing the computation where row i of $\mathbf{C}$ is obtained as a linear combination of the rows of $\boldsymbol{B}$,

$$\mathbf{C}(i,:) = \sum_{\{j | (i,j) \in \mathcal{S}(\boldsymbol{A})\}} \boldsymbol{A}(i,j)\, \boldsymbol{B}(j,:)\,.$$

The set of index pairs (i, j), $c_{ij} \neq 0$ for the ith row of matrix $\mathbf{C}$ is obtained as

$$\{(i,j) \,|\, j \in \mathcal{J}_i\},$$

where

$$\mathcal{J}_i = \bigcup_{\{j | (i,j) \in \mathcal{S}(\boldsymbol{A})\}} \mathcal{S}(\boldsymbol{B}(j,:))\,.$$

```
DETERMINESPSTY($wb$, $colind$, $rowPtrB$, $colindB$)
 1  $nnzci \leftarrow$ size($colind$)
 2  $cp \leftarrow 1$;
 3  for $ip \leftarrow 1$ to $nnzci$
 4      do
 5          $j \leftarrow colind(ip)$;
 6          for $kp \leftarrow rowPtrB(j)$ to $rowPtrB(j+1)-1$
 7              do
 8                  $k \leftarrow colindB(kp)$;
 9                  if $wb(k) = 0$
10                    then
11                        $ind(cp) \leftarrow k$;
12                        $cp \leftarrow cp + 1$;
13                        $wb(k) \leftarrow 1$;
14  for $jp \leftarrow 1$to $cp - 1$
15      do
16          $j \leftarrow ind(jp)$;
17          $wb(j) \leftarrow 0$;
18  return $ind$
```

(a)

$$\begin{pmatrix} 2 & 3 & 5 \end{pmatrix} \Longleftarrow \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

(b)

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \end{array}$$
$$\begin{pmatrix} & \bullet & & & \\ \times & & & \times & \\ & & \bullet & & \bullet \\ & \bullet & & & \bullet \\ \times & & & \times & \times \end{pmatrix}$$

FIGURE 4 | Sparsity calculation.

If the computation $C \leftarrow AB$ is a frequently executed operation where sparsity pattern of matrices A and B do not change, the sparsity pattern of matrix C can be precomputed once and used repeatedly. DETERMINESPSTY determines the sparsity pattern of vector $c = a^\top B$ where vector a and matrix B are sparse and are specified by arrays *colind*, and *rowPtrB* and *colindB*, respectively; *wb* is a binary work array initialized to zero. Figure 4 demonstrates algorithm DETERMINESPSTY. Array *colind* associated with the sparsity pattern of the sparse row vector shown in Figure 4(b) is

1	3	4

such that only the corresponding rows of matrix B are accessed. The elements in these rows are marked with •. Work vector *wb* is used to mark the column corresponding to a nonzero entry while the column index itself is recorded in the output array *ind* in Lines 3 through 13 of DETERMINESPSTY. After processing all the marked rows, *wb* is reset to zero in Lines 14 through 17. Once the sparsity pattern of matrix $C = AB$ is determined the CSR data structure for C can be set up and DETERMINESPSTY can be modified to calculate the actual nonzero values.

Sparse Gauss Elimination

Solving a linear system $Ax = b$ is usually approached by computing a decomposition of A:

$$PAQ = LU,$$

where P and Q are permutation matrices, L is unit lower triangular, and U is upper triangular. In addition to maintaining numerical stability, it is important to ensure that the factors L and U retain most of the exploitable sparsity or structure of matrix A. The vector x that solves the linear system can be obtained as

$$\text{Solve for } y : Ly = Pb; \text{ Solve for } z : Uz = y;$$
$$\text{Obtain } x : x \leftarrow Qz.$$

Classical Gaussian elimination transforms the coefficient matrix into an upper triangular form by systematically 'zeroing' selected entries of the matrix and updating entries that are linearly related to the zeroed entries. Let

$$v^\top = \begin{pmatrix} v_1 & \cdots & v_k & v_{k+1} & \cdots & v_n \end{pmatrix}$$

be a vector where for some $k < n$, $v_k \neq 0$. For $k = 1$ we let $\tau^\top = (\tau_1 \tau_2 \cdots \tau_n)$, where $\tau_1 = 0$, $\tau_i = v_i/v_k, i = 2, \ldots, n$, and define $M_1 = I - \tau e_1^\top$. Then,

$$M_1 (:, 1)^\top = (1 - \tau_2 \cdots - \tau_n)$$
$$\text{and } M_1 (:, j) = e_j, j = 2, \ldots, n.$$

Now, $\hat{v} = M_1 v$ will have $\widehat{v}_1 = 1$ and $\widehat{v}_i = -\tau_i v_1 + v_i = -(v_i/v_1) v_1 + v_i = 0, i = 2, \ldots, n$. In general, the multiplication matrix $M_k, k = 1, 2, \ldots, n-1$ will be

$$M_k = I - \tau e_k^\top = \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\tau_{k+1} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\tau_n & 0 & \cdots & 1 \end{pmatrix},$$

where $\tau_1 = \tau_2 = \cdots = \tau_k = 0$ and $\tau_i, i = k+1, \ldots, n$ will be determined from the vector to be multiplied. Thus, the classical Gaussian elimination of matrix A is achieved by the action of the multiplication matrices on the successive columns of matrix A in $n-1$ steps:

$$U = M_{n-1} M_{n-2} \cdots M_2 M_1 A \quad \text{implying}$$
$$A = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1} U,$$

where $M_k^{-1} = I + \tau e_k^\top$ which can be verified directly. Importantly, the product $L = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}$ is unit lower triangular where

$$L\left(k, k+1 : n\right) = -M_k\left(k, k+1 : n\right),$$

giving the desired form $A=LU$. In a computer implementation, though, one would never form matrix M_k explicitly and then multiply. In fact, the memory cells storing the elements of A are overwritten:

$$A \leftarrow L - I + U$$

$$= \begin{pmatrix} U(1,1) & U(1,2) & \cdots & U(1,k) & \cdots & U(1,n) \\ \tau^{(1)}(2) & U(2,2) & \cdots & U(2,k) & \cdots & U(2,n) \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ \tau^{(1)}(k) & \tau^{(2)}(k) & \cdots & U(k,k) & \cdots & U(k,n) \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ \tau^{(1)}(n) & \tau^{(2)}(n) & \cdots & \tau^{(k)}(n) & \cdots & U(n,n) \end{pmatrix}.$$

Let $A^{(1)} \equiv A$, and for $k = 2, 3, \ldots, n$, $A^{(k)} = M_{k-1}A^{(k-1)}$.

Assume that we have just finished step $k-1$ and computed

$$A^{(k)} = \begin{pmatrix} A_{11} & A_{12} \\ z^\top & a^\top \\ 0 & A_{22} \end{pmatrix},$$

where $A_{11} \in \mathbb{R}^{k-1 \times k-1}$ is upper triangular, $A_{12} \in \mathbb{R}^{k-1 \times n-k+1}$, and $z \in \mathbb{R}^{k-1}$ is identically zero, $a \in \mathbb{R}^{n-k+1}$, and $A_{22} \in \mathbb{R}^{n-k \times n-k+1}$. Then,

$$A^{(k+1)} = M_k A^{(k)} = \left(I - \tau e_k^\top\right) A^{(k)}$$
$$= A^{(k)} - \tau A^{(k)}(k,:).$$

Now,

$$\tau A^{(k)}(k,:) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \tau_{k+1} \\ \vdots \\ \tau_n \end{pmatrix} \begin{pmatrix} 0 & \cdots & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \end{pmatrix},$$

so that the only elements of $A^{(k)}$ that are updated are $A^{(k)}(k+1:n, k+1:n)$:

```
1  for j ← $k+1$ to $n$
2      do
3          $A^{(k+1)}(j, j:n) \leftarrow A^{(k)}(j, j:n) - \tau_{k+1} A^{(k)}$
           $(j-1, j+1:n)$
```

with $n-k$ SPAXPY operations of the form $y \leftarrow \alpha x + y$.

Thus far we have taken the permutation matrices P and Q to be the identity matrix. From a computer implementation point of view a sparse or structured computation typically proceeds by first analyzing the structure of the input sparse data so as to gather information that will be utilized in setting up data structures for the actual computation. As will be demonstrated later, (e.g., in the determination of sparse Jacobian matrices) great savings in computation are achieved by first preprocessing the sparsity into a form that can be utilized in the actual numerical calculations. From a software construction view point it is desirable to separate the two phases cleanly. The two main issues in Gaussian elimination algorithms are: stability of the Gauss transformation and the minimization of fill-in entries in the factors.[18] The algorithm breaks down when a component of vector $\boldsymbol{\tau}$ involves a division by a numerical zero; the algorithm is unstable if a denominator is very small in magnitude relative to its numerator (see Ref 19 and the references therein). Except for a few special cases such as symmetric positive-definite or diagonally dominant[19] systems it is almost always necessary to reorder rows and columns, i.e. find permutations P (for row permutation) and Q (for column permutation) such that the division by smaller quantities is minimized as much as possible in the reordered matrix PAQ—an operation commonly known as *pivoting*. Similarly, it has been frequently observed that there are permutations P and Q such that the reordered matrix PAQ yields fewer fill-ins in its triangular factors L and U compared with that of matrix A. A classic example is the ‘arrow-head’ matrix shown in Figure 5. The left pattern leads to complete fill-ins in the factors L and U. To see this, observe that the first column of the multiplier matrix M_1 contains only nonzero entries. As the components of the first row of the arrow-head matrix are all nonzero, the SPAXPY updates create fill-ins in every other row. On the other hand, by exchanging columns 1 and 5, and rows 1 and 5, one obtains the pattern displayed on the right in Figure 5. Assuming that pivoting is not required it can be verified that there will be no fill-ins.

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & 0 & 0 & 0 \\ \times & 0 & \times & 0 & 0 \\ \times & 0 & 0 & \times & 0 \\ \times & 0 & 0 & 0 & \times \end{pmatrix} \qquad \begin{pmatrix} \times & 0 & 0 & 0 & \times \\ 0 & \times & 0 & 0 & \times \\ 0 & 0 & \times & 0 & \times \\ 0 & 0 & 0 & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}$$

FIGURE 5 | Effect of permutation on fill-in: original pattern (left), permuted pattern (right).

Unfortunately, in general, it is not possible to perform pivoting to enable stability of the Gaussian transform which also minimizes fill-ins; each task may require dissimilar permutations of the columns and rows.[8,18] Algorithm GE-PARTIALP presents Gaussian elimination with *partial pivoting* ($Q \equiv I$).

```
GE-PartialP($A$)
1  for $j \leftarrow 1$ to $n-1$
2      do
3         Find $k \geq j$ such that $|A(k,j)| = \max\{|A(i,j)|, i = j, \ldots, n\}$;
4         Swap $\{A(j,:), A(k,:)\}$;
5         $A(j+1:n, j) \leftarrow \frac{1}{A(j,j)} A(j+1:n, j)$;
6         for $l \leftarrow j+1$ to $n$
7             do
8                $A(l, j+1:n) \leftarrow A(l, j+1:n) - \frac{A(l,j)}{A(j,j)} A(j, j+1:n)$;
```

Line 4 exchanges the current row and the pivot row. Line 5 determines the jth column of $L - I$ and Lines 6–8 perform the row updates on the submatrix

$$A\left(j+1:n, j+1:n\right).$$

At the conclusion of algorithm GE-PartialP the input matrix A is replaced with $PA = LU$. The pivoting strategy in Line 3 is rarely helpful in reducing fill-ins.[8] The original sparsity pattern gets altered in performing the update in Line 8:

1 If $A(l,j) = 0$ or $A(j,j') = 0, j' \in \{j+1, \ldots, n\}$, then $A(l)$ does not change.
2 If both $A(l,j)$ and $A(j,j'), j' \in \{j+1, \ldots, n\}$, are nonzero, then each $A(l,j') = 0 \in \{j+1, \ldots, n\}$ turns into a nonzero, creating a fill-in entry.

Figure 6 displays the submatrix $A(j:n, j:n)$ before (on the left) and after (on the right) the outer product update of Line 8—the jth column and the jth row of A are overwritten with corresponding elements of the factors. The left part of the figure shows the sparsity pattern of an arbitrary row which is updated according to the code in Line 8. As $A(j+1,j) = 0$, entries $A(j+1, j+1:n)$ remain unchanged. However, $A(l,j) \neq 0$ and $A(j,j') \neq 0$ imply that $A(l,j')$ which was originally zero now turned into a nonzero value—a fill-in. We refer to Ref 19 for an introductory discussion regarding different pivoting strategies and to Ref 9 for practical considerations in computer implementations.

A simple but effective strategy employed in software implementation to prevent excessive fill-in while limiting the growth of the nonzero elements of the factors is to use some form of Markowitz condition.[20] The Spaxpy operation in GE-PartialP indicates that the number of fill-in entries produced by the outer-product update depends on the number of nonzero entries in the pivot row and the pivot column, i.e. sparser the pivot row and column fewer the potential fill-in. Further, sparser the pivot row, fewer the number of multiplications performed in outer-product updates. Concerning the growth of the nonzero elements during the outer-product updates one would select a pivot among the elements satisfying some *threshold*:

$$|a_{ij}^{(k)}| \geq \mu \max_l |a_{lj}^{(k)}|, \tag{1}$$

or

$$|a_{ij}^{(k)}| \geq \mu \max_l |a_{il}^{(k)}|, \tag{2}$$

where $\mu \in (0, 1]$ is a user-supplied parameter. Combining numerical stability with sparsity, the Markowitz criterion is to select as pivot an element

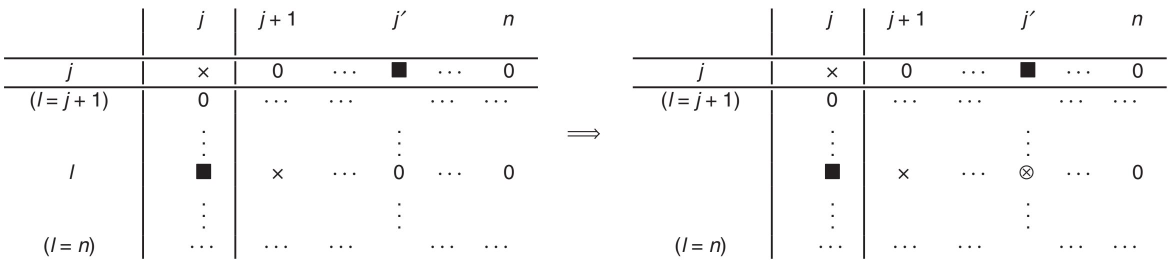


FIGURE 6 | Effect of outer product updates on sparsity.

from among the elements satisfying Eq. (1) or (2) that minimizes

$$(\rho_i - 1)\left(\kappa_j - 1\right),$$

where ρ_i and κ_j denote, respectively, the number of nonzero entries in row i and column j, for element $a_{ij} \neq 0$. In an efficient implementation the rows and columns of the submatrix at the kth step are searched in nondecreasing, nonzero count and an entry is chosen as pivot that satisfies, for example, Eq. (1). Duff et al.[8] discuss schemes for efficient computer implementation of Markowitz criterion.

A very useful sparse technique for general sparsity pattern is to find permutations that reorder the matrix rows and columns to a computationally beneficial structured form called *block-triangular form (BTF)*. Solving a linear system $\boldsymbol{Ax} = \boldsymbol{b}$ where $\boldsymbol{A}$ is block-triangular (lower) can proceed as

$$\text{for } \; l \leftarrow 1, 2, \ldots, p$$

$$\text{solve} : A_{ll}x_{(l)} = b_{(l)} - \sum_{k=1}^{l-1} A_{lk}x_{(k)},$$

$$\text{where } \boldsymbol{A} = \begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & \vdots & \ddots & \\ A_{1p} & A_{2p} & \cdots & A_{pp} \end{pmatrix},$$

and $x_{(k)}$ and $b_{(l)}$ are the corresponding sections of vectors $\boldsymbol{x}$ and $\boldsymbol{b}$, respectively. Prior to computing the BTF of a sparse matrix, one determines permutations $\boldsymbol{P}_1$ and $\boldsymbol{Q}_1$ such that $\boldsymbol{P}_1\boldsymbol{A}\boldsymbol{Q}_1$ has nonzero diagonal.[b] In this phase, an attempt is made to permute and scale the matrix so as to move large nonzero entries to the diagonal. In the second stage, a symmetric permutation $\boldsymbol{P}_2$ is to be found so that $\boldsymbol{P}_2\boldsymbol{P}_1\boldsymbol{A}\boldsymbol{Q}_1\boldsymbol{P}_2 \equiv \boldsymbol{PAQ}$ is block triangular; $p = 1$ corresponds to the unmodified matrix and $p = n$ implies a triangular matrix. Thus, larger the value of p greater the savings. An efficient algorithm for finding BTF is given in Ref 21. The algorithm is based on depth-first search of a directed graph associated with matrix $\boldsymbol{A}$ and runs in $O(n + nnz)$.

SOLVING NONLINEAR SYSTEM OF EQUATIONS

In this section we are concerned with solving a system of nonlinear equations:

Given $\boldsymbol{F} : \mathbb{R}^n \mapsto \mathbb{R}^n$ find $\boldsymbol{x}^* \in \mathbb{R}^n$ such that

$$\boldsymbol{F}\left(\boldsymbol{x}^*\right) = 0, \tag{3}$$

where $\boldsymbol{F}$ is continuously differentiable.

Newton's Method

Newton's method, in one form or the other, is one of the most widely used and highly effective techniques in optimization. There is a large volume of scientific literature on this method (and its many variants); some of the classic texts are Refs 22–24. For an algorithmic introduction and implementation issues we refer to Ref 25. In the following, we deliberately keep our discussion of Newton's method for solving Eq. (3) short and simple. Meza[26] provides a concise and gentle introduction to this important method and the paper nicely complements the material in this section.

Newton's method for solving Eq. (3) can be derived by considering a linear approximation to $\boldsymbol{F}$ at the current iterate $\boldsymbol{x}^{(k)}$:

$$\boldsymbol{F}\left(\boldsymbol{x}^{(k)} + \boldsymbol{s}\right) \approx \boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) + \boldsymbol{F}'\left(\boldsymbol{x}^{(k)}\right)\boldsymbol{s}, \tag{4}$$

where the element in row i and column j of the Jacobian matrix of $\boldsymbol{F}$ at $\boldsymbol{x}$, $(\boldsymbol{F}'(\boldsymbol{x}))(i,j)$, is the partial derivative of component i of vector $\boldsymbol{F}$ with respect to $\boldsymbol{x}(j)$. The Newton equations (5)

$$\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) + \boldsymbol{F}'\left(\boldsymbol{x}^{(k)}\right)\boldsymbol{s} = 0$$

$$\text{yielding } \; \boldsymbol{F}'\left(\boldsymbol{x}^{(k)}\right)\boldsymbol{s} = -\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) \tag{5}$$

can be solved for step $\boldsymbol{s}$. Updating the current iterate gives

$$\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \boldsymbol{s}, \tag{6}$$

which is a better approximation to $\boldsymbol{x}^*$ than $\boldsymbol{x}^{(k)}$. Given a starting guess $\boldsymbol{x}^{(0)}$ $(k = 0)$ to $\boldsymbol{x}^*$, Newton's algorithm for solving Eq. (3) can be specified as an iterative procedure:

```
NewtonMethod($\boldsymbol{F}, \boldsymbol{x}^{(0)}$)
1  repeat Compute $\boldsymbol{F}'(\boldsymbol{x}^{(k)})$
2         Solve $\boldsymbol{F}'(\boldsymbol{x}^{(k)})\boldsymbol{s}^{(k)} = -\boldsymbol{F}(\boldsymbol{x}^{(k)})$
3         Set $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \boldsymbol{s}^{(k)}$
4         Set $k \leftarrow k + 1$
5  until $\boldsymbol{x}^{(k)}$ is a "good enough" approximation to $\boldsymbol{x}^*$.
```

If $\boldsymbol{x}^* \in \mathbb{R}^n$ exists such that $\boldsymbol{F}(\boldsymbol{x}^*) = 0$ and given a good starting guess, Newton's method is quadratically convergent (see e.g.,) if the Jacobian matrix at the solution $\boldsymbol{x}^*$ is nonsingular and $\boldsymbol{F}$ posses Lipschitz continuous first derivative in some neighborhood of $\boldsymbol{x}^*$. On the negative side, Newton's method requires, at each iteration, the determination of the Jacobian matrix at the current iterate—a computationally prohibitive step when n is large. Further, the linear system solve in Step 2 can be ill-conditioned or even singular thereby posing algorithmic difficulties.

Fortunately, many large-scale problems exhibit useful structure,[27–30] e.g. the components of $\boldsymbol{F}$ (nonlinearly) depend on a very small subset of the n independent variables.[28] The computation or approximation of the Jacobian matrix of such problems as well as the associated linear solve can be made economical by exploiting these structural information. The bulk of the computational work in NewtonMethod is concerned with the determination of the Jacobian matrix at $\boldsymbol{x}^{(k)}$ and solving the linear system at each iterative step of the **repeat** loop. If a computer code implementing function $\boldsymbol{F}$ is available, one may write code for analytic derivatives by hand. For large or complicated functions this process is time-consuming and highly error-prone. A suitable alternative to hand-coded derivatives is to approximate the Jacobian by a finite-difference scheme. A more recent approach is to employ automatic or algorithmic differentiation (AD) (see Ref 31 for a comprehensive treatment of AD techniques and their software implementation) to determine the nonzero entries of the Jacobian up to the machine precision at a slightly higher computational cost than the finite-difference approximation. In either method, great savings in computation can be achieved if the sparsity structure of the Jacobian is known a priori or can be computed easily,[32] and does not change from iteration to iteration.

Let the matrix $\boldsymbol{A}$ in Figure 1 be the Jacobian matrix of some function $\boldsymbol{F}$ at $\boldsymbol{x}$. We have,

$$\frac{1}{\epsilon}\left(\boldsymbol{F}\left(\boldsymbol{x}+\epsilon\left(\mathrm{e}_1+\mathrm{e}_2+\mathrm{e}_3\right)\right)-\boldsymbol{F}(\boldsymbol{x})\right)^{\top} \approx \left(\widetilde{a}_{12} \quad \widetilde{a}_{21} \quad \widetilde{a}_{33} \quad \widetilde{a}_{42} \quad \widetilde{a}_{51}\right), \tag{7}$$

where $\epsilon > 0$ is a small increment (step-size), and $(\tilde{\cdot})$ indicates that the entry is an approximation to the true value. Therefore, matrix $\boldsymbol{A}$ can be approximated with only three extra function evaluations of the form $\boldsymbol{F}(\boldsymbol{x}+\epsilon\boldsymbol{s})$ for direction $\boldsymbol{s}$ set to $\mathrm{e}_1+\mathrm{e}_2+\mathrm{e}_3$, e_4, and e_5 in succession, in addition to evaluating $\boldsymbol{F}$ at $\boldsymbol{x}$. A complicating factor in finite-difference method is the contradictory nature of the two sources of error. In general, a small ϵ in Eq. (7) implies a small truncation error. On the other hand, in finite-precision arithmetic it leads to cancelation error (because of subtraction of nearly equal floating point numbers) potentially magnified by the division by small ϵ (see, e.g., Chapter 5 of Ref 23). With AD technology there is no step size to choose, and the entries of $\boldsymbol{A}$ are obtained as three forward evaluations of AD forward mode in the form of products $\boldsymbol{As}$. The computed values are free of truncation error (of FD approximations) and are obtained at a small constant ($\sim$3) multiple (see Chapter 4 of Ref 31) of the cost of evaluating the function $\boldsymbol{F}$. The central point here is that if the sparsity structure of a group of columns is such that for every pair of column indices $j \neq l$ in the group $\mathcal{S}(\boldsymbol{A}(:,j)) \cap \mathcal{S}(\boldsymbol{A}(:,l))$ is empty, then only one AD forward accumulation or one extra function evaluation will suffice to determine the nonzero entries in those columns. In other words, columns $\boldsymbol{A}(:,j)$ and $\boldsymbol{A}(:,l)$ are *structurally orthogonal*, i.e. there is no index i for which $a_{ij} \neq 0$ and $a_{il} \neq 0$. Columns 1–3 of the example matrix $\boldsymbol{A}$ in Figure 1 are structurally orthogonal. The nonzero entries of matrix $\boldsymbol{A}$ can be determined from

$$\boldsymbol{AS} = \boldsymbol{B}, \tag{8}$$

with

$$\boldsymbol{S} = \left(\mathrm{e}_1+\mathrm{e}_2+\mathrm{e}_3 \quad \mathrm{e}_4 \quad \mathrm{e}_5\right),$$

$$\text{and} \quad \boldsymbol{B} = \begin{pmatrix} \widetilde{a}_{12} & 0 & 0 \\ \widetilde{a}_{21} & \widetilde{a}_{24} & 0 \\ \widetilde{a}_{33} & 0 & \widetilde{a}_{35} \\ \widetilde{a}_{42} & 0 & \widetilde{a}_{45} \\ \widetilde{a}_{51} & \widetilde{a}_{54} & \widetilde{a}_{55} \end{pmatrix},$$

where matrix $\boldsymbol{B}$ is obtained, for example, via FD (or AD forward accumulation). Thus, the columns of matrix $\boldsymbol{B}$ correspond to the directional derivatives of function $\boldsymbol{F}$ in the directions $\boldsymbol{S}(:,j), j = 1, 2, 3$; with FD each directional derivative costs one extra evaluation of function $\boldsymbol{F}$ while with AD it is a small multiple of the cost of evaluating the function $\boldsymbol{F}$.

```
CPR-Ordering($\mathcal{S}(\boldsymbol{A})$, $group$, $ngroup$)
1  $ngroup \leftarrow 1$
2  for $j \leftarrow 1$ to $n$
3      do
4         let $\mathcal{L} = \{l \mid \boldsymbol{A}(:,j)$ is not structurally orthogonal to $\boldsymbol{A}(:,l)\}$ and
5            $c_m = \min\{c \mid c \in \{1,\ldots,ngroup+1\} \neq group(l), l \in \mathcal{L}\}$
6         $group(j) \leftarrow c_m$
7         if $c_m > ngroup$
8           then
9                $ngroup \leftarrow ngroup+1$
```

Exploiting known sparsity as noted above is Curtis, Powell, and Reid.[4] Algorithm CPR-ORDERING takes as input the sparsity pattern of matrix A and an array, *group*, initialized to zero. The index set $\mathcal{L}$ in Line 4 represents columns that cannot be grouped with column j; c_m represents the least-numbered structurally orthogonal column group that can include column j. After the algorithm is executed, column j belongs to the group *group*(j), and *ngroup* holds the number of structurally orthogonal groups. It is straight forward to set up an explicit sparse data structure for matrix S of Eq. (8) using CSC scheme: allocate arrays *gptr* of length $ngroup+1$ and *gcolind* of length n such that columns belonging to structurally orthogonal group k is obtained as $gcolind(i), i=gptr(k)$ to $gptr(k+1)-1$.

We now illustrate, in procedure FD-SPJS, a simple pseudocode implementation for obtaining an approximation to the nonzero entries of the directional derivative

$$F'(x)\,S(:,k) \equiv B(:,k)$$

(of function $F:\mathbb{R}^n \mapsto \mathbb{R}^n$) corresponding to structurally orthogonal group k.

```
FD-SPJS($gptr, gcolind, k, \eta, B$)
1  $w \leftarrow$ FD-SPJD($F, x, \eta, gptr, gcolind, k$);
2  for $ind \leftarrow gptr(k)$ to $gptr(k+1)-1$
3      do
4          $j \leftarrow gcolind(ind)$;
5          for each $i$ where $F'(i,j) \neq 0$
6              do
7                  $B(i,k) \leftarrow \frac{w(i)}{\eta(j)}$;
```

FD-SPJD is a user-defined function to compute the difference $F(\mathbf{x}+\eta)-F(\mathbf{x})$, where the array η initially containing zeros can be defined to contain the finite-difference increments $\epsilon(j)$ corresponding to columns j in structurally orthogonal group k:

```
1  for $ind \leftarrow gptr(k)$ to $gptr(k+1)-1$
2      do
3          $j \leftarrow gcolind(ind)$;
4          $\eta(j) \leftarrow \epsilon(j)$;
```

One of the main goals of grouping algorithms is to minimize the number of groups. Unfortunately, this minimization problem is NP-Hard.[33,34] Furthermore, the order in which the columns are scanned in CPR-ORDERING affects the number of structurally orthogonal groups the columns can be partitioned into. Graph-theoretic formulations of the problem[33,35,36] have been found to be advantageous leading to a number of effective heuristics and software implementations.[37–39]

Some Classical Iterative Methods for General Sparse Matrices

Iterative methods for solving linear system can be broadly categorized into *relaxation* methods and the *projection* methods. The relaxation methods are the oldest and well understood,[40,41] and have an iterative scheme of the form

$$Mx^{(k+1)} \leftarrow b + Nx^{(k)},$$

and equivalently,

$$x^{(k+1)} \leftarrow M^{-1}b + M^{-1}Nx^{(k)}$$
$$\text{giving } x^{(k+1)} \leftarrow Tx^{(k)} + c, \tag{9}$$

where $A=M-N$ is a splitting of matrix A in the linear system $Ax=b$. In classical Jacobi (Ja) and Gauss–Seidal (GS) the splitting is determined as $M_{\text{Ja}}=D, N_{\text{Ja}}=(E+F)$ and $M_{\text{GS}}=D-E, N_{\text{GS}}=F$, respectively; matrix A is partitioned as $A=D-E-F$ with $D=\text{diag}(A(i,i), i=1,\ldots,n)$ is a diagonal matrix, $-E$ is the strictly lower triangular part of A with zeros elsewhere, and $-F$ is the strictly upper triangular part of A with zeros elsewhere. Denoting the true solution by x^* and defining the error in the kth iterate $e^{(k)}=x^{(k)}-x^*$, we have

$$e^{(k+1)} = Te^{(k)} \text{ implying } \|e^{(k+1)}\| \leq \|T\|\|e^{(k)}\|,$$

where $\|T\| = \max_{x\neq 0} \|Tx\|/\|x\|$ for any vector norm $\|\cdot\|$. Thus, for any initial guess $x^{(0)}$ the iteration will converge to the solution x^* if the norm of the iteration matrix is less than 1.

Inexact Newton Methods

The quadratic convergence of Newton's method is local and it can be argued that the linear system $J(x^{(k)})s^{(k)} = -F(x^{(k)})$ be solved only approximately when the current iterate is far from the solution x^*. For large-scale problems it may be computationally prohibitive to determine the Jacobian and solve the linear system exactly. Dembo et al.[42] define *inexact Newton methods*:

```
INEXACTNEWTONMETHOD($F, x^{(0)}$)
1  repeat
2        Choose some $\eta_k \in [0,1)$ to determine $s_k$ satisfying:
             $\|F'(x^{(k)})s_k + F(x^{(k)})\| \le \eta_k \|F(x^{(k)})\|$
3        Set $x^{(k+1)} \leftarrow x^{(k)} + s^{(k)}$
4        Set $k \leftarrow k+1$
5     until $x^{(k)}$ is an "acceptable" approximation to $x^*$.
```

The local convergence of inexact Newton methods is established by the forcing term η_k:

Theorem 1. [42] *Assume that* $\eta_k \le \eta_{\max} < t < 1$. *There exists* $\epsilon > 0$ *such that, if* $\| x^{(0)} - x^* \| \le \epsilon$, *then the sequence of inexact Newton iterates* $\{x^{(k)}\}$ *converges to* x^*. *Moreover, the convergence is linear in the sense that*

$$\|x^{(k+1)} - x^*\|_* \le t\|x^{(k)} - x^*\|_*.$$

Further, if the sequence of iterates $\{x^{(k)}\}$ *converges to* x^*, *then* $\{x^{(k)}\}$ *converges to* x^* *super linearly if and only if*,

$$\|F'\left(x^{(k)}\right)s_k + F\left(x^{(k)}\right)\| = O\left(\|F\left(x^{(k)}\right)\|\right).$$

In the algorithm INEXACTNEWTONMETHOD we can solve approximately for s_k in

$$F'(x_k)\,s_k = -F(x_k)$$

so that the relative residual in the linear system is sufficiently small. If the residual is $r_k = F'(x_k)s_k + F(x_k)$ then in INEXACTNEWTONMETHOD the relative residual will be $\| r_k \| \le \eta_k \| F(x_k) \|$. If the Newton equations are solved using an iterative solver, e.g. Krylov subspace method we have a *Newton–Krylov method*. The choice for the forcing sequence can be made so as to enhance the convergence rate and enable robust computer implementation.[43] There are strategies to attain global convergence based on, e.g. trust-regions or backtracking.[44,45]

SUMMARY

We have presented a birds-eye-view of some of the fundamental sparse matrix techniques relevant in solving systems of nonlinear equations. Structured and sparse problems are ubiquitous in areas of science and engineering where accuracy of the model output is directly proportional to the discretization of the governing differential equations—finer the discretization, larger the associated linear or nonlinear system, and smaller the model error, in general. Although we have focused on a very small selection of techniques (and mostly related to our own research), pointers to pertinent literature are provided throughout the paper for more in-depth look. Our presentation has tried to reflect the intricate nature of the interplay of algorithmic techniques underlying the computational methods and effective data structures suitable for complex computing systems. While it is almost impossible to give an exhaustive list of problems that give rise to computations involving sparse or structured matrices, in the following we identify topics that we have omitted in this paper but are highly relevant and important.

A very important class of iterative methods is the Krylov subspace methods. We refer to the excellent texts[7,46] for a definitive treatment of the modern iterative methods. For a symmetric positive definite matrix the triangular factorization can be done stably (in that the permutation matrices for numerical stability are simply the identity matrix) by the Cholesky factorization $A = LL^{\top}$. George and Liu[47] give an in-depth description of the methods for sparse Cholesky and Davis[9] covers the more recent developments.

Inverse problems can sometimes be formulated as nonlinear least-squares problems and display significant exploitable sparsity.[48] The KKT system that needs to be solved for the Newton's correction involves a coefficient matrix that contains sparse blocks. Employing an iterative method requires multiplication of the Jacobian (sensitivity) matrix by a vector. Since the Jacobian is dense, it is never formed explicitly.

Large-scale linear programming problems[49] often are inherently sparse and structured. Many software tools routinely employ sparse linear algebra in implementing simplex method. Unfortunately, simplex method in the worst-case may be inefficient. Interior-point methods, radically different from simplex method, are guaranteed to run in polynomial time, and are especially suitable for very large-scale problems.

NOTES

[a] Some authors use *static* and *dynamic* for read-only and read–write, respectively.
[b] If this cannot be done, then the matrix A must be structurally singular which implies numerical singularity.

ACKNOWLEDGMENTS

The authors thank the referees for excellent comments and suggestions. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Research Council of Norway (NFR).

REFERENCES

1. Gilbert JR, Moler C, Schreiber R. Sparse matrices in Matlab: design and implementation. *SIAM J Matrix Anal Appl* 1992, 13:333–356.
2. Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach*. 5th ed. San Francisco, CA: Morgan Kaufmann Publishers Inc.; 2011.
3. Duff IS, Heroux MA, Pozo R. An overview of the sparse basic linear algebra subprograms: the new standard from the BLAS technical forum. *ACM Trans Math Softw* 2002, 28:239–267.
4. Curtis AR, Powell MJD, Reid JK. On the estimation of sparse Jacobian matrices. *J Inst Math Appl* 1974, 13:117–120.
5. Golub GH, Van Loan CF. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press; 1996.
6. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, Van der Vorst H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1994.
7. Saad Y. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2003.
8. Duff IS, Erisman AM, Reid JK. *Direct Methods for Sparse Matrices*. New York: Oxford University Press Inc.; 1986.
9. Davis TA. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2006.
10. Filippone S, Buttari A. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM Trans Math Softw* 2012, 38:23:1–23:20.
11. Boisvert RF, Pozo R, Remington KA. The matrix market exchange formats: Initial design. Technical Report 5935, National Institute of Standards and Technology, Gaithersburg, MD, 1996.
12. Duff IS, Grimes RG, Lewis JG. Sparse matrix test problems. *ACM Trans Math Softw* 1989, 15:1–14.
13. Gundersen G, Steihaug T. Data structures in Java for matrix computations. *Concurr Comput Pract Exper* 2004, 16:799–815.
14. Haque SA, Hossain S, Maza MM. Cache friendly sparse matrix-vector multiplication. In: Moreno Maza M, Roch J-L, eds. *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. New York: ACM; 2010, 175–176.
15. Pinar A, Heath MT. Improving performance of sparse matrix-vector multiplication. In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing '99*. New York: ACM; 1999.
16. Yzelman AN, Bisseling RH. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM J Sci Comput* 2009, 31:3128–3154.
17. Nishtala R, Vuduc RW, Demmel JW, Yelick KA. When cache blocking of sparse matrix vector multiply works and why. *Appl Algebra Eng Commun Comput* 2007, 18:297–311.
18. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2002.
19. Higham NJ. Gaussian elimination. *WIREs Comput Stat* 2011, 3:230–238.
20. Markowitz HM. The elimination form of the inverse and its application to linear programming. *Manage Sci* 1957, 3:255–269.
21. Duff IS, Reid JK. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Trans Math Softw* 1978, 4:137–147.
22. Bertsekas DP. *Nonlinear Programming*. Belmont, MA: Athena Scientific; 1999.
23. Dennis JE Jr, Schnabel RB. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations (Classics in Applied Mathematics, 16)*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1996.
24. Nocedal J, Wright SJ. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. 2nd ed. New York: Springer; 2006.
25. Kelley CT. *Solving Nonlinear Equations with Newton's Method. Fundamentals of Algorithms*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM); 2003.
26. Meza JC. Newton's method. *WIREs Comput Stat* 2011, 3:75–78.
27. Coleman TF, Xu W. Fast (structured) Newton computations. *SIAM J Sci Comput* 2008, 31:1175–1191.
28. Griewank A, Toint PL. On the unconstrained optimization of partially separable functions. In: Powell

MJD, ed. *Nonlinear Optimization 1981*. New York: Academic Press; 1982, 301–312.

29. Jackson RHF, McCormick GP. The polyadic structure of factorable function tensors with applications to high-order minimization techniques. *J Opt Theory Appl* 1986, 51:63–94.
30. McCormick GP, Sofer A. Optimization with unary functions. *Math Prog* 1991, 52:167–178.
31. Griewank A, Walther A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2008.
32. Griewank A, Mitev C. Detecting Jacobian sparsity patterns by Bayesian probing. *Math Prog* 2002, 93:1–25.
33. Coleman TF, Moré JJ. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J Num Anal* 1983, 20:187–209.
34. Hossain S, Steihaug T. Optimal direct determination of sparse Jacobian matrices. *Optim Methods Softw* 2012. doi: 10.1080/10556788.2012.693927.
35. Gebremedhin AH, Manne F, Pothen A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Rev* 2005, 47:629–705.
36. Hossain S, Steihaug T. Graph models and their efficient implementation for sparse Jacobian matrix determination. *Disc Appl Math* 2013, 161:1747–1754.
37. Coleman TF, Garbow BS, Moré JJ. Software for estimating sparse Jacobian matrices. *ACM Trans Math Softw* 1984, 10:329–345.
38. Gebremedhin AH, Patwary MMA, Nguyen D, Pothen A. Colpack: software for graph coloring and related problems in scientific computing. Technical report, Purdue University, OH, 2011.
39. Hasan M, Hossain S, Steihaug T. DSJM: a software toolkit for direct determination of sparse Jacobian matrices. *SIAM Workshop on Combinatorial Scientific Computing (CSC09)*, 2009, 2.
40. Varga RS. *Matrix Iterative Analysis*. Springer Series in Computational Mathematics, vol. 27. 2nd ed. New York: Springer-Verlag New York; 2009.
41. Young DM. *Iterative Solution of Large Linear Systems*. New York, NY: Dover; 2003(Unabridged republication of the 1971 Edition. New York/London: Academic Press; MR 305568).
42. Dembo RS, Eisenstat SC, Steihaug T. Inexact Newton methods. *SIAM J Num Anal* 1982, 19:400–408.
43. Pernice M, Walker HF. Nitsol: a Newton iterative solver for nonlinear systems. *SIAM J Sci Comput* 1998, 19:302–318.
44. Pawlowski RP, Shadid JN, Simonis JP, Walker HF. Globalization techniques for Newton-Krylov methods and applications to the fully coupled solution of the Navier-Stokes equations. *SIAM Rev* 2006, 48:700–721.
45. Pawlowski RP, Simonis JP, Walker HF, Shadid JN. Inexact Newton dogleg methods. *SIAM J Numer Anal* 2008, 46:2112–2132.
46. Greenbaum A. *Iterative Methods for Solving Linear Systems*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1997.
47. George A, Liu JW. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice Hall; 1981.
48. Vogel CR. *Computational Methods for Inverse Problems*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2002.
49. Vanderbei R. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science, vol. 196. 4th ed. New York: Springer-Verlag New York; 2013.

FURTHER READING

Axelsson O. *Iterative Solution Methods*. New York: Cambridge University Press; 1994.

Coleman TF. *Large Sparse Numerical Optimization*. New York: Springer-Verlag New York, Inc.; 1984.

Kepner J, Gilbert J, eds. *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2011.

Ortega JM, Rheinboldt WC. *Iterative Solution of Nonlinear Equations in Several Variables*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2000.